Systematic Composition of Independent Language Features

Arvid Butting[a,*], Robert Eikermann[a], Oliver Kautz[a], Bernhard Rumpe[a], Andreas Wortmann[a]

[a]*Software Engineering, RWTH Aachen University, Aachen, Germany*

Abstract

Systematic reuse is crucial to efficiently engineer and deploy software languages to software experts and domain experts alike. But "software languages are software too", and hence their engineering, customization, and reuse are subject to similar challenges. To this effect, we propose an approach for composing independent, grammar-based language syntax modules in a structured way that realizes a separation of concerns among the participants in the life cycle of the languages. We present a refined concept of systematic and controlled syntactic variability of extensible software language product lines through identification of syntax variation points and derivation of variants from independently developed features. This facilitates reuse of software languages and reduces the efforts of engineering and customizing languages for specific domains. We realized our concept with the MontiCore language workbench and assessed it through a case study on architecture description languages. Ultimately, systematic and controlled software language reuse reduces the effort of software language engineering and fosters the applicability of software languages.

1. Introduction

Model-driven development (MDD) [2, 88] leverages (domain-specific) software languages to reduce the conceptual gap between problem domain challenges and the software engineering solutions [26, 80]. Thus, efficient engineering, customization, and reuse of software languages has become a prime concern in MDD and gave rise to the field of software language engineering (SLE) [46, 35]. SLE develops methods and techniques to engineer domain-specific modeling languages (DSMLs) within language workbenches. But "software languages are software too" [23] and as such are subject to the same challenges regarding engineering, customization, and reuse as other software. Consequently, research in SLE has produced a variety of solutions to engineer languages based on metamodels or grammars, interpreters or generators, well-formedness rules in metalanguages or programming languages. Metamodels [7, 41, 58, 72, 90, 94] describe the abstract syntax (*i.e.,* structure) of languages as graphs of associated classes without providing a concrete syntax enabling instantiation of models. Grammars also describe the structure of a language, but can support an integrated definition of concrete syntax as well [22, 47, 50, 67]. From these grammars, model processing infrastructure (*e.g.,* a parser that translates textual models into abstract syntax instances) can be derived automatically, which greatly facilitates the efficient usage of DSMLs.

Efficient reuse is crucial to the success of software engineering [34] and software languages [13]: language users and language engineers can greatly benefit from reusing common, established, and mature language concepts. For software language engineers, reuse reduces the effort in engineering new languages from scratch. For language users, reusing concepts reduces the effort required to comprehend a language. For instance, Java reuses many concepts of C++, which lowers the barrier of using Java for C++ developers. With the digitalization of all aspects of our lives, more and more domain experts (mechanical engineers, physicist, lawyers, *etc.*) become software developers to some degree. They must be able to reify their domain expertise in software, which can be integrated with other systems. For some domains and challenges, such as software architectures in general [54] or Industry 4.0 in particular [91], many specific languages have been developed already and even more are under development.

Software product line engineering has produced methods and means to capture variability and commonalities for increasing software reuse, *e.g.,* within feature models [14]. A software product line describes several variants of software in an integrated fashion to mitigate cloning and owning of the commonalities in independent software projects. Software product line engineering techniques have been applied to software language engineering to form language product lines in several approaches [56].

We present a refined concept of controlled language variability based on reusable definitions of language syntaxes as modules within language components that can be composed to produce new languages from established building blocks. This facilitates a posteriori extensibility with additional language concepts, supports concrete syntax, and enables (re-)using languages (parts) without explicitly foreseeing this usage at the respective languages' design time. To enable a systematic reuse

[☆]This research has partly received funding from the German Federal Ministry for Education and Research under grant no. 01IS16043P. The responsibility for the content of this publication is with the authors.

[*]Corresponding author

Email addresses: butting@se-rwth.de (Arvid Butting), eikermann@se-rwth.de (Robert Eikermann), kautz@se-rwth.de (Oliver Kautz), rumpe@se-rwth.de (Bernhard Rumpe), wortmann@se-rwth.de (Andreas Wortmann)

[BEK+19] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. Wortmann:
Systematic Composition of Independent Language Features.
In: Journal of Systems and Software, 152, pp. 50-69, 2019.
www.se-rwth.de/publications/

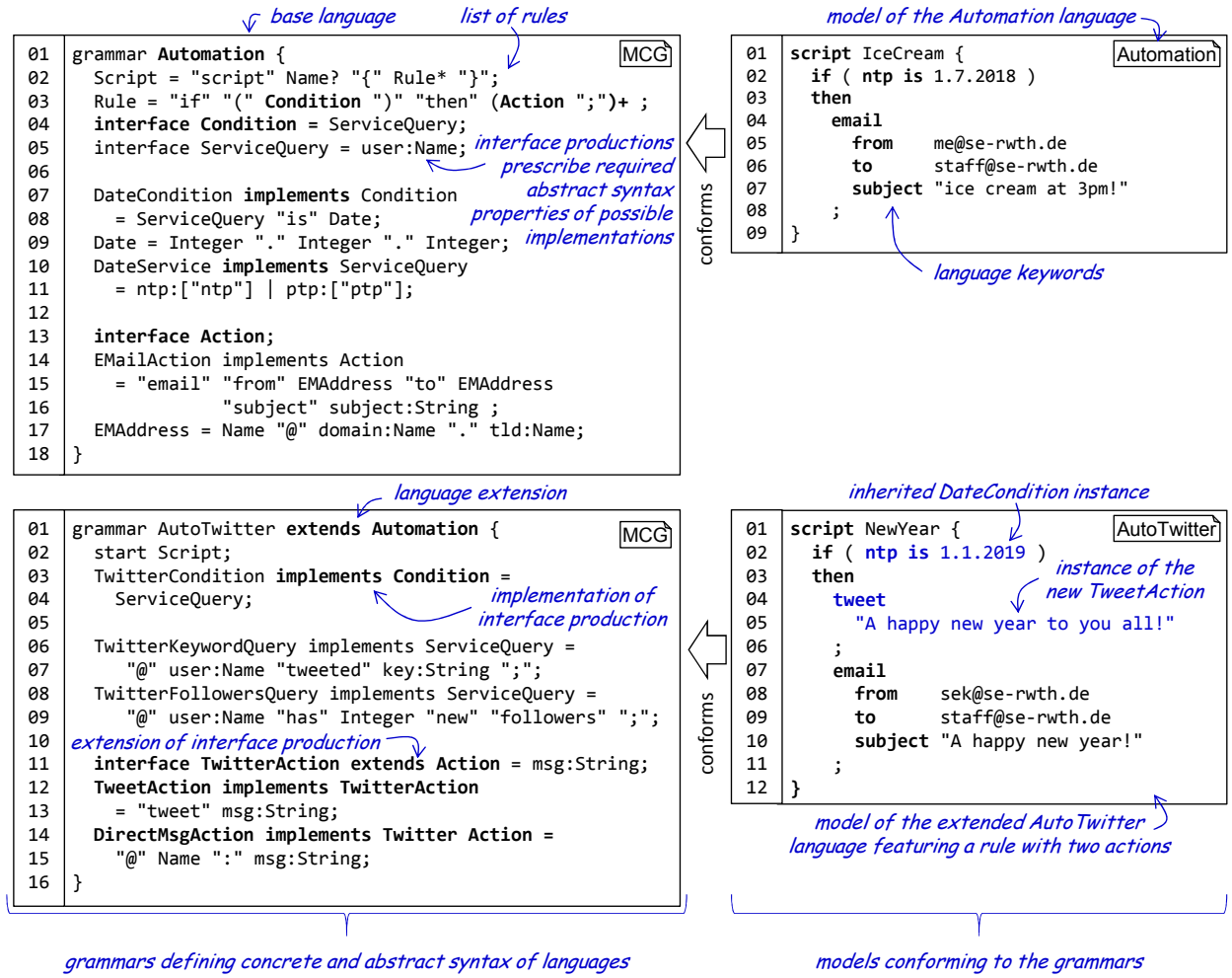


Figure 1: The `AutoTwitter` grammar (bottom left) extends the `Automation` grammar (top left) and implements its extension points (*e.g.,* interface `Condition`) to enable interacting with Twitter. Conforming models are depicted on the right.

that can ensure syntactically valid language variants, we propose explicating this variability in form of language product lines. Concrete language variants of the product line are derived from composing the incorporated language components and language-processing tooling for these can be derived on push-button basis. The key ideas of our approach are:

- to enable compositional language development by decomposing languages into composable language components,
- to enable automated language derivation via composing language components,
- to increase reuse of concrete and abstract syntax as well as tooling (*e.g.,* well-formedness check implementations) via the composable language components, and
- to decouple language development from language composition and language variant derivation.

Our concept is realized using the MontiCore [67] language workbench and leverages its composition mechanisms based on well-defined language extension points. This enables a *controlled* language composition supporting validation on product line level. The individual languages can be independent of each other, which enables these to be developed by different language engineers.

This paper is an extended version of [8], in which the basic principles of composable language components have been introduced, including a concept for systematic language reuse, its MontiCore realization, and a case study on architecture description languages. The contributions of the extended version include:

- An example product line for an imperative language.
- A concept for controlled abstract syntax extension through interface productions.
- Integration with FeatureIDE [74].
- Extended description of the involved stakeholders and more detailed discussion of related work.

In the remainder, Section 2 presents an example of an extensible base language for imperative event-based programming

and its configuration based on an explicit variability model. Afterwards, Section 3 introduces necessary preliminaries. Section 4 details our concept for systematic and controlled language reuse. Section 5 describes its realization, before Section 6 presents a comprehensive case study on architecture description languages. Thereafter, Section 7 debates observations and Section 8 discusses related work. Section 9 concludes.

2. Example

Consider engineering a modeling language for software automation that is supposed to be tailored to application-specific requirements by other developers. This language should provide general concepts and a framework for specifying automation rules consisting of extensible trigger conditions and extensible actions that are executed if the trigger condition holds. With models of this language, users can describe system-level rules (such as sending an email if a specific drive is full) as well as service-crossing rules (*e.g.,* whenever someone tweets about the hashtag `#SLE`, attach it to a file and save it locally).

Engineering such a language raises two requirements:

1. The base language must be extensible to support easy integration of new conditions and actions (*e.g.,* automatically check a weather web service on your phone to show whether you need an umbrella).

2. The extension must be restrictable to prevent interacting with undesirable trigger conditions and rules (*e.g.,* accessing a root drive of the server).

Regarding extensibility, we leverage controlled underspecification in the base language's abstract syntax (*i.e.,* the structure of a language) to enable a-posteriori integration of new language elements in a restricted fashion. Consequently, integration of new language elements should be possible in predefined places of the abstract syntax only and requires that the new language elements fulfill specific properties. The MontiCore [30, 67] grammar `Automation`, depicted in Figure 1 (top), illustrates this. It defines both concrete syntax (*i.e.,* the appearance of models of the language) and abstract syntax of conforming models and yields extension points that enable implementation and extension by inheriting grammars. The technical necessities for this are explained in Section 3.

Models conforming to this grammar are named scripts consisting of a list of rules. Hence, the grammar's start production `Script` (l. 2) begins with the concrete syntax keyword `script`, followed by an optional (denoted by "?") name and a block comprising a list of `Rule` instances. Rules (l. 3) begin with the keyword `if`, followed by a `Condition`, the keyword `then` and at least one (denoted by "+") `Action`. Both, `Condition` (l. 4) and `Action` (l. 13) are *interface* productions [67]. Other grammar productions can implement interface productions, which means that they can be applied in derivation wherever the interface production is expected on a right-hand side of another production. Technically, interface products are eliminated before handing the grammar to a parser generator. This

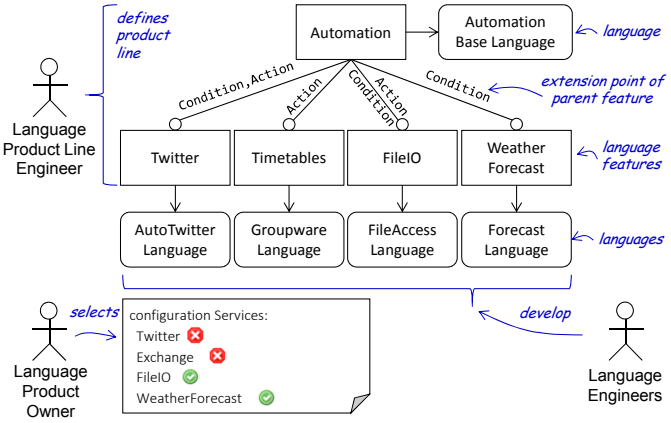


Figure 2: Example of a language product line using the `Automation` language a basis for language products featuring rules on twitter interaction, calendar services, file access, and weather forecasts.

is done by introducing an ordinary production with a left-hand side that is equal to the left-hand side of the interface production and a right-hand side that defines an alternative between all productions implementing the interface production. For example, before handing the `Automation` grammar (*cf.* Figure 1) to a parser generator, the interface production `Condition` is eliminated via adding a production `Condition = DateCondition`. Here, the right-hand side solely uses the nonterminal `DateCondition`, because the `DateCondition` production is the only production that implements the `Condition` interface production. Interface productions are translated into interface classes of the abstract syntax. Interface productions can prescribe (on their right-hand sides) abstract syntax properties required by possible implementations. For instance, `Condition` requires that every implementation yields a `ServiceQuery` (l. 5), which is another interface that requires implementations to yield a `Name`. The `Automation` base language enables conditions over dates (ll. 7-11) and actions to send emails (ll. 13-17). If additional language features are desired, `Automation` must be extended properly.

With MontiCore [30, 67], language extension can have the form of grammar inheritance between a base grammar and an inheriting grammar, *i.e.,* the inheriting grammar inherits production rules from the base grammar, such as the `AutoTwitter` grammar depicted at bottom left of Figure 1. The inheriting grammar imports all productions of the inheriting grammar and can optionally override or extend these. The `AutoTwitter` grammar reuses the start production of its parent grammar (l. 2) and defines the production `TwitterCondition` (ll. 3-4) as an implementation of `Condition`. To this end, the `TwitterCondition` introduces the concrete syntax keyword `@`, which is followed by a name and a `ServiceQuery`. The latter is necessary as it is required by the interface `Condition` of `Automation`. Moreover, the grammar introduces the two implementations `TwitterKeywordQuery` and `TwitterFollowersQuery` of `Automation`'s interface `ServiceQuery`. The grammar also extends the interface production `Action` of the `Automation` grammar with the interface production `TwitterAction`, which

prescribes that twitter actions always yield at least a `String` message. Moreover, `AutoTwitter` introduces two implementations of `TwitterAction` (and, hence, `Action`) that enable tweeting and sending direct messages.

Models conforming to both grammars are depicted in Figure 1 to their respective right. The script `IceCream` (top right) uses a date condition and an email action to describe a rule informing staff on the first of July about ice cream. To this end, it uses elements of the `Automation` grammar only. The script `NewYear` (bottom right) conforms to the `AutoTwitter` grammar and hence can use its elements as well as the elements inherited from `Automation`. The model, hence, can use a date condition of `Automation` (l. 2), a tweet action (ll. 4-6), and an email action (ll. 7-11) to send out New Year's greetings via email and twitter. Leveraging this form of grammar inheritance enables opportunistic reuse and extension, but lacks control regarding the languages that are combined, *i.e.,* which services are made available to the users. Thus, restricting the kinds of services that rules can interact with is impossible. Also, it requires comprehensive understanding of software language engineering concepts (such as grammar inheritance and interface productions) and that the inheriting languages are aware of inherited concepts (*e.g.,* the interfaces of their parent grammars). The latter also avoids reusing a language inheriting from a base language in a context different from the base language,

To enable controlled reuse, liberate domain experts from becoming language experts, and facilitate language composition, we leverage the concepts of software product lines: Available trigger conditions and actions should be provided in form of independent features arranged into a feature diagram by a language product line engineer. After ensuring the validity of feature combinations, the responsible product line engineer can use this to configure language products by selecting which features to include. Based on this selection, for instance, a concrete `Automation` language product can be derived that contains only the trigger conditions and actions desired by the application context. The language modules in the features are independent of a concrete context and can be reused for different language product lines. For the `Automation` language, a possible language product line is depicted in Figure 2. Here, the product line engineer (*e.g.,* the service provider) establishes the language product line by relating the generally available language modules in features in a meaningful way. To this end, she ensures that the available feature combinations lead to meaningful languages with respect to their syntax (*e.g.,* prevent duplicate keywords). This requires language engineering expertise and comprehending the available features, but liberates the product line engineers and the modelers from this. Each of the related features contains a complete language that can be independent of other languages, *e.g.,* of the base language. Through establishing the language product line, the engineer also defines which features implement which extension points of their parent feature (or of the base language) in the language product line. Through this, she ensures that, for instance, no actions implement the `Condition` interface. Here, the feature `Twitter` contributes conditions and actions, whereas the feature `WeatherForecast` contributes conditions only.

Based on a feature configuration, we derive a specific language product comprising only the selected features. Throughout the next sections, we describe how to decouple language components of each other such that they can be developed independently, *i.e.,* how to mitigate the inheritance relationship between a base language and an extension. Furthermore, we describe how to explicate extension points of language components, how to compose independent language components while being able to reuse tooling for a language component, and how to separate the concerns involved in the process of developing and using language product lines.

3. Preliminaries

Languages, in general, are characterized as "the set of sentences" [12, 46] that constitute the language, which also applies to software languages. As software languages – compared to natural languages – typically have a simpler structure to be processable by machines, this definition can be refined. This also is necessary for these languages to be better accessible to investigation. A common refinement [28, 31] is that DSMLs comprise

- a *concrete syntax*, describing the sentences of the language, which are build from words (textual languages), diagram elements (graphical language), or other representations,
- a (minimal) *abstract syntax*, describing the (essential) structure of sentences of the languages,
- a *semantic domain* (typically a well-defined mathematical theory) that can express the meaning of sentences, and
- a *semantic mapping*, which gives each well-defined sentence a meaning within the semantic domain.

The definition of the abstract syntax typically uses either metamodels (such as with EMF Ecore [73] or MPS [87]) or grammars (such as Neverlang [76] or Xtext [5]). The concrete syntax of textual models is usually also defined by grammars [67], whereas the concrete syntax of graphical models is usually defined via graphical or projectional editors, *e.g.,* with Sirius [82]. Textual models are usually parsed to translate the concrete syntax into abstract syntax. Whereas some approaches do not allow to create models that are not well-formed through robust projectional editors [71], most approaches require explicit checks to ensure well-formedness.

The language workbench MontiCore [67, 48] leverages extended context-free grammars (CFGs) for the integrated definition of concrete and abstract syntax [30] of DSMLs. From a CFG, MontiCore generates the corresponding (Java) abstract syntax classes, a parser for models of the language based on ANTLR [62], a model checking infrastructure that facilitates developing well-formedness rules realized in Java, and a model-to-text code generation infrastructure based upon the FreeMarker [25] template engine [1].

The generated parsers translate textual models into instances of the abstract syntax classes, the abstract syntax trees (ASTs), which are processed by handcrafted well-formedness rules registered with the generated model checking infrastructure that is realized with the visitor pattern [27]. Well-formed models are processed further and can, ultimately, be transformed into arbitrary target language artifacts by employing code generators. To validate well-formedness constraints not expressible with CFGs, MontiCore features compositional *context conditions* (CoCos). The target language artifacts then can realize the DSLs semantics and be subject to further analyses.

An exemplary MontiCore grammar `ADLGrammar` is depicted in Figure 3 (top left). It describes the quintessential elements of an ADL [55], *i.e.,* components that yield interfaces of typed ports and subcomponents that exchange messages through connectors between their ports (ll. 2-4). Nonterminals in MontiCore grammars start with an upper case letter and terminals are surrounded by quotation marks. The right-hand sides of grammar productions contain references to other nonterminals and terminals and use cardinalities ('?','*','+') known from regular expressions. To distinguish references to (non)terminals on the right-hand side of a grammar production, they can optionally be named. For instance, the reference to the nonterminal `Name` in l. 5 is named `id`. Besides 'usual' grammar productions, MontiCore supports abstract productions and interface productions. These do not directly influence the parser, *i.e.,* they cannot be derived. Instead, they influence the abstract syntax: interface productions translate to interfaces of the abstract syntax data structure and abstract productions are translated into abstract classes. Interface productions can prescribe required abstract syntax elements of possible implementations on their right-hand side. The interface production `Port` (l. 5), for instance, prescribes the presence of exactly one nonterminal `Name` with the name `id` and the presence of a `Type`. This mechanism enables underspecification of the ordering of prescribed abstract syntax elements as well as potential further (non)terminals in any rule that implements the interface production. Interface productions and abstract productions can be used on any right-hand side in the same way as normal grammar productions. A grammar production implementing an interface production, *e.g.,* the `EncryptedPort` implementing the `Port`, has to provide the required abstract syntax elements of the interface productions in the prescribed cardinality. If this is not the case, MontiCore detects it and aborts generation of language-processing tooling. In the abstract syntax this is reflected by the abstract syntax class (*e.g.,* the class EncryptedPort) implementing the abstract syntax interface (*e.g.,* the interface Port). During parsing, a production implementing an interface production can be applied at any point where the interface production is expected. For example, an `EncryptedPort` can be part of a component, as it implements the `Port`. Interface productions and abstract productions are translated into ordinary productions before handing the grammar to a parser generator. The `interface` and `abstract` modifiers in MontiCore grammars, however, effect the modifiers of the generated abstract syntax classes: Interface productions are translated to (Java) interfaces and abstract productions are translated to abstract (Java) classes. Detailed documentation about MontiCore is available [67].

MontiCore also supports compositional language integration via extension, embedding, and aggregation [30, 67]. Through extension of one or more parent grammars, a grammar inherits the grammar rules and terminal of its parent grammar(s) and can extend and override these. This, for instance, enables to eliminate rules, introduce new alternatives for inherited interfaces, or extend individual inherited rules. From inheriting grammars, MontiCore produces refined AST classes that inherit from the AST classes of the extended or overridden rules.

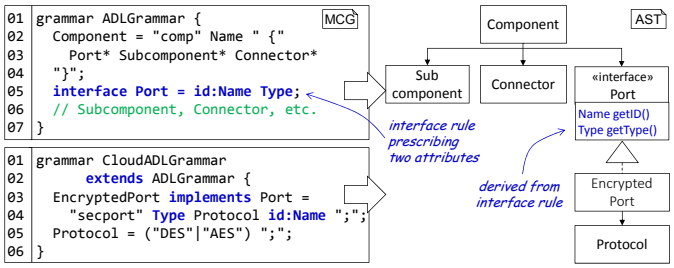


Figure 3: Example of grammar inheritance in MontiCore.

Software product line engineering (SPLE) conceives methods to handle similar software products in an integrated fashion within software product lines. Each software product, also referred to as variant, of the product line has certain commonalities and differences with regard to other variants. Developing commonalities independently is usually costly, time-consuming, error-prone, and subject to co-evolution of the independent parts. The aim of SPLE, therefore, is a reduction of developing commonalities independent of each other, but rather to reuse commonalities across the variants. Feature diagrams [3, 14, 44] group common parts of software within features. Features are arranged in feature trees. In our approach, we employ feature diagrams with the usual relations (mandatory, optional) between parent features and sub features as well as feature groups (alternative, exclusive). Besides this, the possible feature configurations can be restricted via cross-tree constraints (requires, excludes). Based on the restrictions in the feature model, a selection of a set of features determines a certain variant. The selection can be validated according to the constraints of the feature model.

4. Modeling Language Variability

In software engineering in general, and in component-based software engineering in particular, bundling of software into reusable components has proven to be helpful [39, 59]. Component-based software engineering pursues the idea of "black-box" components that can be reused off-the-shelf. This requires independent components with explicit interfaces for their composition. As software languages are software too [23], this also applies to software languages. Consequently, modularization and variability of software languages are investigated as well. Previous work [36, 67, 68, 86] conceived concepts and methods to facilitate composition of software languages.

For instance, the MontiArc [10] architecture description language [55] composes a host component & connector language with an embedded automata language and embedded statements of a Java DSL, and it is aggregated [30] with the UML/P [66] class diagram (CD) language to enable architectures with behavior models operating in the context of a CD. From applying this language and its variants to various domains including cloud systems [60], automotive [29], and robotics [33], we identified the following research questions that a concept for software language variability should solve:

RQ1: *How can variability of the syntax within a language (module) be realized?*
Answering this question requires a dedicated notion of variability that applies to the syntax of languages. One possibility to realize variability in general is the usage of underspecification. The question should answer how elements of a language syntax that are expected at a certain place can be properly underspecified. While underspecification inherently abstracts from expected elements, a certain typing of such elements is desirable for composition of syntaxes.

RQ2: *How to compose independent language modules?*
Composition integrates the syntax of two language modules. There needs to be additional information on which exact parts of the syntaxes, *i.e.,* which extension and which extension point, should be composed. This is typically realized via "glue". The solution to this question, therefore, requires a notion for this glue and dedicated composition operators for the constituents of a language module. Furthermore, proper handling of ambiguities that can arise in the composition of two language syntaxes must be assured.

RQ3: *How can language composition be guided to obtain meaningful combinations of language modules?*
To control, which language modules can be combined in a meaningful way, we explicate certain combinations of language components within families of similar languages. Modeling a family of similar languages requires selecting suitable modeling techniques to reflect the constituents realizing a language (solution space) as well as to represent the variability using appropriate techniques (problem space). Apart from this, an approach for guided combinations requires a notion of validity of the composition, *i.e.,* a member of the family of languages.

RQ4: *How can development of language modules and their composition be decoupled?*
In practice, revealing all technical details of language modules for their composition complicates their reuse and requires the language engineer composing the modules to have detailed knowledge about these. Instead, it should be sufficient to communicate the underlying conceptual model of each individually developed language module, so that the composition could be maintained by another person. This person can arrange language modules such that only meaningful compositions are allowed. This alleviates a domain engineer who selects a subset of these meaningful compositions of modules from requiring software language engineering expertise.

In the remainder, Section 4.1 presents a concept to model variability of a single language component. Subsequently, Section 4.2 explains a concept for modeling variability of multiple language components and for engineering language product lines. Then, Section 4.3 presents the roles involved in developing and using a language product line.

4.1. Reusable Language Components

The basis for our proposed variability mechanism are *language components*, which are language modules comprising (1) a grammar providing an integrated description of abstract syntax and concrete syntax; (2) a (possibly empty) set of dedicated grammar productions acting as extension points of the abstract syntax; and (3) a set of well-formedness rules. A language component is an independent, standalone *model* that is not aware of being used with an underlying variability model. To foster maturing in the sense of component-based software engineering [59], language components should be developed independent of each other, to be reusable in different contexts whenever possible. The aim of this is to enable reusing language components "off-the-shelf" by only requiring limited knowledge on the intricacies of a language component. We use grammars as foundational description mechanism for language syntaxes and show how to describe extension points and extensions of language syntax through grammars. The notion of extension points in the abstract syntax, however, can be transferred to metamodel-based syntax definitions as well.

We explicate the extension points within language components to denote the reduced effort of understanding the intricacies of a grammar when a language component is reused. For instance, in the exemplary language product line in Figure 2, the language component `AutomationBaseLanguage` can be used mainly by communicating the extension points `Condition` and `Action`. The language product line engineer has to understand implementation details on other grammar productions (*e.g.,* `EMAddress`) only on a more abstract level, unless the composition is invalid, *e.g.,* due to ambiguities. A further advantage of explicating the extension points is that language engineers who develop language components can hide certain parts of the language that must not be extended as, *e.g.,* known from private methods in Java. In general context-free grammars, our notion of extension points within a language's syntax applies to arbitrary grammar productions. This means any grammar production can serve as extension point. A grammar production used as extension point can provide a default implementation (that is its right-hand side). In context-free grammars in general, all productions must provide a right-hand side and with these, all extension points must provide a default implementation. Variability is realized through underspecification of alternatives to the default implementation. In other words, extending an extension point is realized by adding the extension as alternative to the right-hand side of the extension point. To foster only meaningful combinations of languages and enable reusing a language component based on knowing its extension points, language engineers have to decide carefully which grammar productions should act as extension points.

We prescribe that the cardinality of an extension point is only realized via the variability model, and not via the language component itself. For example, the grammar should not restrict that an extension point must be extended at all, once, or multiple times. This increases the reusability of a language component within different contexts. An extension is a grammar production that provides syntactical elements to resolve the underspecification of an extension point. To this effect, extensions provide the right-hand side that was underspecified in an extension point. Typically, the extension is contained in a different grammar than the extension point it implements. More precisely, connecting two grammars via an extension point and an extension is realized within two steps: (1) Merge all productions of both grammars to form a new grammar. The start production of the grammar containing the extension point becomes the start production of the new grammar. (2) Add the left-hand side of the production of the extension as alternative to the right-hand side of the extension point production.

In the realization of the concept, leveraging the powerful language composition techniques of MontiCore, we represent grammar extension points with *interface productions* (*cf.* Section 3). These have the additional advantage, that a concrete right-hand side can be underspecified (*i.e.,* no default implementation is required), but assumptions on the abstract syntax of possible productions implementing the interfaces can be made. This enables controlled extension and reasoning over possible implementations already on the interface level. In this sense, interface productions resemble interfaces or abstract classes in metamodels (depending on the formalism). Consequently, for instance, abstract classes in a metamodel would also enable to represent (controlled) extension points. The concept of language components realizes **RQ1**.

In MontiCore, language components are not required to provide default implementations of their extension points. MontiCore enables a grammar to define an interface production without providing a production implementing the interface production. However, the components themselves are abstract in the sense that it will not be possible to concretize all possible sentences (models) of the language. Nevertheless, language components may provide certain default implementations by providing grammar productions that implement the interface production (or classes implementing the interfaces in metamodel-based approaches). For example, a Statechart language component with an extension point for guard conditions on transitions can optionally include built-in default expressions but still be extensible with additional kinds of expressions (*e.g.,* OCL, LTL, ...) through different language components.

However, explicating extension points via special grammar constructs requires that language engineers have to foresee each extension point during development of a language component. The realization of our concept in MontiCore, for instance, considers all interface productions as extension points, which liberates language engineers in maintaining explicit extension points as separate artifact(s).

Besides the context-free grammar, a language component contains a set of well-formedness rules. These may check the well-formedness of any abstract syntax element of the grammar of the enclosing language component. Where the abstract syntax is arranged as a tree, well-formedness rules should check the most specific syntax element that is possible. This increases the reusability of the well-formedness rule if only parts of the language's syntax are reused. Additionally, it reduces side effects of the well-formedness rule that arise if it is applied for post-hoc added language constructs for which it was not designed to apply for. For example, consider a rule checking the well-formedness of dates (*e.g.,* 30.02.2019 is not considered well-formed) for the `Automation` grammar depicted in Figure 1. The well-formedness rule should be implemented against the abstract syntax element introduced by the nonterminal `Date` as the most specific possible element. If it was implemented against more general elements, such as the `DateCondition` nonterminal, and only the `Date` nonterminal was reused in another context, then the well-formedness rule would not apply there. With MontiCore, well-formedness rules can be implemented against interface productions to assert well-formedness of their right-hand sides. It is possible to check all implemented context conditions via a checking infrastructure. This infrastructure is automatically generated by MontiCore using the grammar [67].

The composition of language components requires different composition operators for the different constituents of a language component. A language component with a grammar and well-formedness rules requires composition operators for these. A language component based on metamodels would require a composition operator for metamodels [17, 19]. Two language components are composed by composing all constituents of these language components.

Multiple language components can be composed at a time, but the composition operator for grammars is directed. Applying all required composition operators produces an integrated, new language component. This integrated language component contains all details necessary to synthesize language-processing tooling for the contained language. Furthermore, it can serve as a language component within different language product lines, *e.g.,* by engineering a new language product line with the generated language component at its base, describing the commonalities of all possible variants.

Our notion of syntactic language components requires composition operators for grammars and for context conditions. The composition operator for grammars takes an ordered list of input grammars and a binding from extensions to extension points (of other grammars). Using these, it embeds extending implementations into the extension points. For MontiCore grammars, this is realized as implementing interface productions, for other context-free grammar by introducing additional alternatives at the extension point. For metamodels, this can be realized by introducing new interface implementations or subclasses of abstract metamodel classes. For grammars, the result would be a composed grammar that includes all grammar productions of the input grammars. For metamodels, this would be a joined metamodel including all concepts and relations of participating metamodels. The composition operator does not resolve potential conflicts, such as, through grammar productions with identical nonterminal names. Instead, this has to be

resolved via additional transformations. All context-free grammars have a single start production that defines the start of the derivation process. The start production of the composed grammar is the start production of the first input grammar[1].

Independent of the abstract syntax realization mechanism, the extension points of the composed language components are the union of extension points of the input language components. In other words, all extension points are preserved in the composed language component.

Similarly, the sets of well-formedness rules are preserved. The composition operator merges these such that the composed language component contains the union of all participating well-formedness rules. Again, the composition operator does not check if well-formedness rules contradict. If this is the case, the set of well-formed models may be empty. For instance, consider a well-formedness rule assuming that automation models (*cf.* Section 2) are well-formed if these contain exactly one action and another rule considering well-formed models to have more than one twitter action. Apart from that, composing sets of well-formedness rules is straightforward. The concept for composing language components (*i.e.,* composing grammars, their extension points, and well-formedness rules) realizes the identified **RQ2**.

4.2. Language Product Lines

With language components and a notion of their composition, new languages can be built up from independent, reusable modules. However, it is cumbersome and error-prone to identify, which language components can be reused for a certain purpose and how their composition is arranged, *i.e.,* which extension points and which extensions are connected.

To remedy this, our approach leverages a variability model to organize language components and establish bindings between these. For the realization, we identified feature models with their usual relations [44] and constraints as a suitable modeling technique. In the feature model of a language product line, each feature of a feature model instantiates exactly one language component. The concept, however, can be adapted and applied to different variability modeling techniques as well. As language components are unaware of being used with features, they can be used with different feature models and even with different features within a single feature model. For instance, two features could instantiate the same language component but embed it differently, which is useful when reusing more generic language components – such as expression languages – with different extension points. Moreover, the loose coupling between feature model and language components facilitates extending the language product line with new features and language components to support their evolution.

Using a feature model, bindings, and language components, a language product line can be constructed (*cf.* **RQ3**). Each possible variant (product) of the language product line is described by a valid feature configuration of the feature model. This prevents uncontrolled composition of language components, *i.e.,* a form of composition that is neither foreseen or nor intended by the language product line engineer and leverages language users from comprehending internals and composition of different language components.

Moreover, properties of extension points, such as being mandatory, optional, or exclusive with another extension point are solely realized via the variability model. This yields the advantage of the language components to be better reusable in different contexts, with different applied extension points, and to support staged configuration [15]. A *requires* relationship between two features (denoting that one feature relies on the presence of another feature), enables to indicate that a language component of one feature must have knowledge about the language component of the other feature. This reflects, *e.g.,* inheritance between the respective grammars or dependencies between the metamodels. It enables modeling that, for instance, a feature providing an expression language requires another feature providing a type system.

A binding between extension point and extension is specific to two features of a concrete feature model and the language components they instantiate. Therefore, we organize all bindings of a language product line within a single dedicated binding model. The model describes for each feature of the feature model which language component it instantiates by stating a mapping from a feature name to a language component name. Additionally, it defines binding rules that connect extension points and extension via their nonterminal names. To this effect, it contains one name mapping per feature of the feature model and one binding rule per edge between two features of the feature model. A binding model is specific to a language product line and its related language components and is the only connection between these, otherwise decoupled, parts as it indicates that a language component is instantiated in each feature it is bound to. Consequently, reusing the feature model of a language product line, *e.g.,* with language components based on another language definition formalism, becomes feasible. Each binding rule constitutes (1) the parent feature and the extension point that the rule applies to and (2) the child feature and the extension that is bound to the extension point.

Uncontrolled composition could occur if the connection between extension point and extension would be based upon, *e.g.,* both grammar productions to have the same name or by "autoconnecting" an extension to all extension points. To mitigate this, we explicate bindings. As the root feature does not have a parent feature, there is no binding rule that binds extensions of the root language feature to other extension points. To this effect, our approach allows extensions to be extension points (therefore in MontiCore, interface productions) as well. Extending an extension point *refines* the extension point and enables concretizing its required abstract syntax elements, as well as to enable more sophisticated structuring of features of the language product line. This is demonstrated by an example in Section 6.

Extending a selected set of features with further features does not invalidate the syntax of models that were valid before, as

[1] This appears to be a limitation at this point, but in combination with a feature model as described in Section 5, the first input grammar is always the one used in the root feature.

adding new language components cannot eliminate syntax elements. This *conservative extension* [67] of the syntax, enables reusing tooling for multiple language variants of the product line. However, it cannot be used to guarantee semantical correctness of analyses. Through adding new language elements, several analyses might produce different results. Consider, for instance, an analysis counting actions defined in a model conforming to the `Automation` grammar (*cf.* Figure 1). If a language extension would add nested actions, these would not be taken into account individually unless the analysis is modified.

Deriving a language variant begins with selecting features to compile a feature configuration. The feature configuration has to be valid with regard to the feature model and its constraints. Then, our language composition tool composes the language components of all selected features at once. As the composition operator is ordered, the order of language components is computed by a depth-first search starting from the root feature. Here, the order of siblings is irrelevant.

Composition produces an integrated grammar, a joined set of context conditions, and all extension points of the input language component are retained. Furthermore, a new language component containing these is generated that can be used, *e.g.,* as root for a new language product line. A language component should contain all information necessary to produce language-processing tooling for the defined language. This activity can be performed on the composed language component as last step of language variant derivation.

Developing a language product line by organizing which language components can be composed does not necessarily require to understand in-depth details of the language components' internals. This facilitates reusing language components that the language product line engineer did not develop on her own. Thus, she only has to consider all details of a language component if side effects of composing the grammars (such as ambiguities in abstract or concrete syntax) arise. To avoid language compositions that cause invalid language variants, she can, for instance, establish cross-feature relations preventing these (*e.g.,* by mutual exclusion of features contributing ambiguous grammars). This fosters a separation of concerns between language engineers developing language components and language product line engineers arranging these meaningfully (*cf.* **RQ4**).

4.3. Roles

Our method to enable modeling language reuse through variability rests on a separation on related concerns along different roles:

- *Language engineers* develop language components that encapsulate abstract syntax and concrete syntax.
- *Language product line engineers* are language engineers that create language families as feature models that describe possible characteristics of the family's language products. To this end, they collect relevant language components, assign these to features, and define how these realize extension points of language components of their parent features.
- *Language product owners* configure the language family to obtain a language product for a specific domain, context, or application. Based on this configuration, the tool chain generates model-processing tooling for the selected product.
- *Modelers* employ the generated modeling processing tooling to analyze models and transform these into GPL artifacts.

The separation of concerns between the participating roles frees the individual roles to be involved in all phases of conceiving language product lines and language products. Figure 4 details the activities of the different stakeholders: Creating a language product line begins with *language engineers* developing the modeling languages that the product line combines. We assume the languages are defined using grammars defining the concrete and abstract syntax and its possible extension points. Extension points of the grammar become extension points of the language component. After the grammar is defined, language engineers define rules describing the well-formedness of models of the language. Grammar and well-formedness rules then are explicated within a language component model.

The *language product line engineer* collects the different language components and arranges these into a feature model describing the product line. To this end, she first defines the desired features within a feature model and models their constraints according to the domain's needs. Then, within the binding model, she instantiates a language component for each feature or the feature model. Afterwards, she creates binding rules for each connection between extension point and extension. In the resulting product line, features denote selectable language components and the language components of child features are available for embedding into extension points of the language components of their parent features. Optionally, language product line engineers can create glue that is specific to the connection between two instances of language components that are connected via the feature model. This glue comprises *e.g.,* well-formedness rules and further analyses or tooling specific to the composition of two languages. Apart from these, it is possible to perform handcrafted adaptations of, *e.g.,* grammars that cause ambiguities when they are composed through creating a grammar that inherits from the original one and overrides conflicting productions.

Based on the language family, the *language product owner* selects features of interest for a specific domain, context, or application and uses the language variability infrastructure to automatically compose grammars and well-formedness rules into new model processing tooling for the specific configuration. With this, the *modelers* can process models using the selected features transparently as if developed for a single monolithic language.

To this end, the separation of concerns among the described roles alleviates experts to have capabilities in all fields of expertise among the process of developing language product lines as depicted in Figure 5. This reflects in the different kinds of artifacts created and used throughout the process. The capability

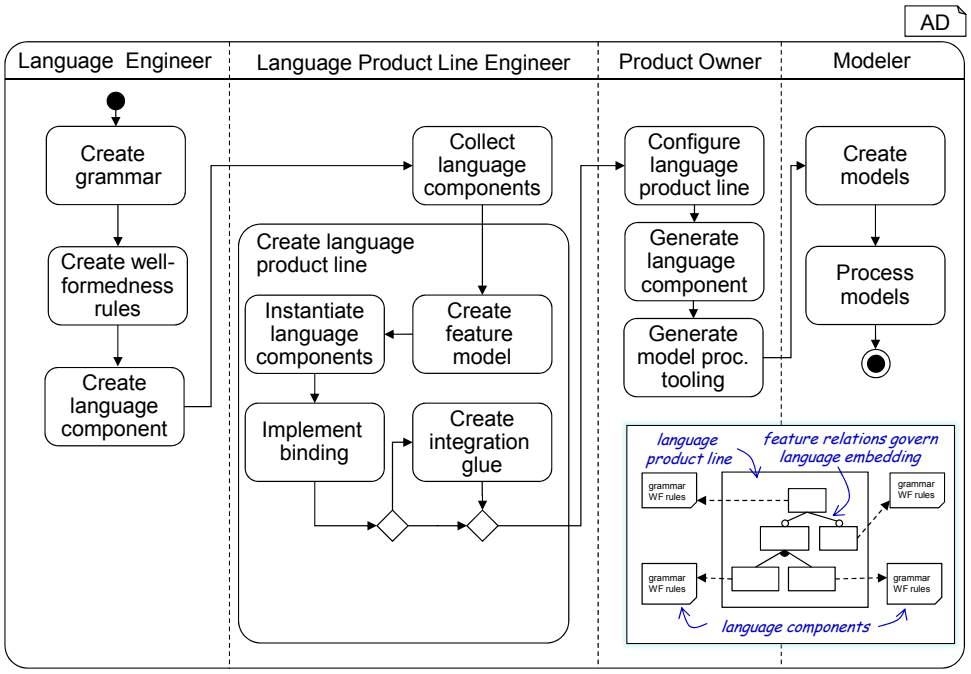


Figure 4: Activities and associated roles participating in creating, configuring, and using language product lines.

C = Create
U = Understand

	Syntax	Well-Formedness Rules	Language Components	Feature Tree	Binding Rules	Feature Configuration
Language Engineer	C	C	C	-	-	-
Language Product Line Engineer	U/C	U/C	U	C	C	-
Language Product Owner	-	-	U	U	-	C
Modeler	(U)	(U)	-	-	-	-

Figure 5: The roles involved in developing language product lines and their required capabilities.

to create artifacts related to syntax and well-formedness of languages is solely required by language engineers, and partially by language product line engineers. The latter only require to create syntax or well-formedness artifacts if conflicts arising in combinations of syntaxes or well-formedness rules result in problems that are to be resolved manually. Language product line engineers, however, have to understand the artifacts realizing syntaxes and well-formedness rules of the involved languages. Modelers are only required to understand these in a more abstract way, *e.g.,* through documentation. Language product owners do not need to understand syntax and well-formedness rules. Language components are created by language engineers and have to be understood by language product line engineers and product owners to create and configure language product lines. Modelers do not have to be aware of the existence of language components. The feature tree is created by the language product line engineer and used by language product owners for configuring language variants through feature configurations. The language product line engineer creates a binding model for each feature tree, in which she connects features to language components and binds extension points of language components with extensions of other language components. Language engineers and modelers do not have to be aware of the presence of feature models, binding rules, and feature configurations.

5. Integrating Languages Syntaxes

This section describes the MontiCore realization of the concept introduced in the previous chapter. The concrete and abstract syntax of a MontiCore [30] language is defined within an extended context-free grammar. Well-formedness checking is realized via context conditions implemented as Java classes (*cf.* Section 3). To define extension points of a grammar, our approach leverages interface grammar productions (interface nonterminals) that the grammar itself can provide default implementations for by containing productions that implement the interface production. As depicted in l. 13 of the top grammar in Figure 1, interface production rules are not required to have a right-hand side. If an interface production has a right-hand side, it indicates nonterminal symbols required to be provided by grammar productions that implement the interface production. Similar to ordinary nonterminals, interfaces can be referenced on the right-hand sides of other production rules (*cf.* `Component` in ll. 2ff). Nonterminals can implement interfaces (ll. 6f), resulting in an abstract syntax structure as depicted by example on the right of Figure 3. Our approach supports composition of *independent* language components. This mitigates the necessity to use grammar composition via inheritance as explained in Section 3 (*cf.* `CloudADLGrammar` in Figure 3), which requires

```
01 language ADLLangComp {                                   LC
02   grammar com.ma.ADLGrammar;
03   cocos {
04     com.ma.cocos.CompNameLowerCase,
05     com.ma.cocos.PortNamesUnique
06   }
07 }
```

Figure 6: The language component ADLLangComp.

```
01 binding for ADL {                                        Binding
02  feature BaseADL uses ADLLangCmp;
03  feature Behavior uses BehaviorLangComp;
04  feature Automaton uses AutomatonLangComp;
05  bind Behavior.BehModel to BaseADL.ADLElement;
06  bind Automaton.IOAutomaton to Behavior.BehModel;
07 }
```

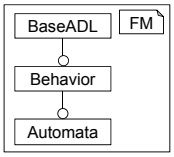


Figure 7: The binding model for the composition of exemplary language components.

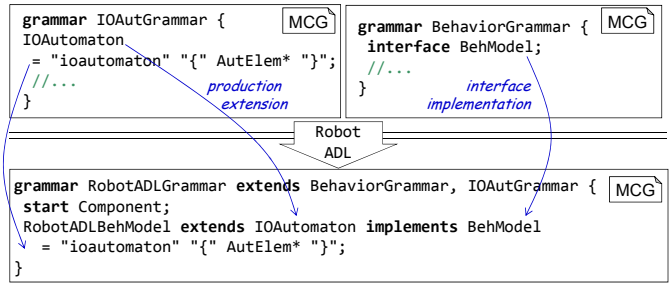


Figure 8: The composition (bottom) of two MontiCore grammars (top).

that the inheriting grammar cannot be used without the inherited grammar and therefore reduces reusability.

Composition of language components realizes the variability in the solution space of our approach. The composition of language syntaxes is performed by composing their context-free grammars and context conditions. MontiCore grammars used in language components can, in general, use grammar inheritance as supported by MontiCore. Grammar inheritance between grammars contained in different language components of a language product line should be performed carefully, as it can yield unintended side effects. Therefore, we recommend indicating such a relationship via a "requires" constraint in the feature model.

Figure 6 depicts a *language component* definition. It holds a reference to a MontiCore grammar and a set of corresponding context conditions. The language component has the name ADLLangCmp, references the ADLGrammar presented in Figure 3, and references two context conditions. All interface nonterminals defined in the referenced grammar are automatically exported as extension points of the language component.

Consider a small language product line for ADLs with a base feature BaseADL, an optional behavior feature, and an optional Automaton feature for the realization of the behavior. The corresponding feature model is depicted in the right of Figure 7. The binding model depicted in the left of Figure 7 contains the relation between features in the feature model (depicted right) and language components (ll. 2-4) that these instantiate. Further, it defines binding rules between extension points and extensions(ll. 5-6). Thess rules define which nonterminals of a child feature are bound to which extension points of a parent feature. For example, in every language variant comprising the Automata feature, the IOAutomaton nonterminal of the grammar contained in the feature's language component is bound to the extension point BehModel exported by the language component of feature Behavior (l. 6). Extension points and extensions have to be identified via names comprising feature and grammar production to be uniquely identifiable within a product line. A single language component, for instance, could be instantiated more than once in a product line. The top of Figure 8 depicts two MontiCore grammars that are part of language components to be composed. The grammar IOAutGrammar is part of the language component AutomatonLangComp and, inter alia, contains the grammar production IOAutomaton. The BehaviorGrammar is part of the language component BehaviorLangComp and provides the interface production BehModel. The grammar RobotADLGrammar depicted at the bottom is the result of the composition of the upper two grammars with the binding model as described above. The relevant part of the binding for this composition is contained in l. 6. The production IOAutomaton of IOAutGrammar uses the extension point (= interface production) BehModel of the grammar BehaviorGrammar. The name of the synthesized grammar is derived from the name of the language variant as prefix and Grammar as suffix. The grammar inheritance mechanism of MontiCore is used to extend both grammars, which enables to reuse all of their grammar productions. For technical reasons, the generated grammar has to reference the start production of the grammar of the root feature, to produce the same top-level abstract syntax element for each generated parser of the product line. For each applied binding rule (= implemented interface production), a new nonterminal is generated in the synthesized grammar. The new production extends the extension nonterminal with extends IOAutomaton and implements the extension point interface with implements BehModel. The effect of implementation of an interface has been explained in Section 3. Extending a production has a similar effect, it can be used in every place of the grammar where the extended production has been applied. With the abstract syntax tree data structure MontiCore generates from a grammar, the corresponding abstract syntax tree classes reflect the same extension and implementation on the level of Java classes and interfaces. Consequently, the abstract syntax class of RobotADLBehModel extends from the abstract syntax class of the IOAutomaton and implements the interface BehModel. This bears the advantage that all algorithms and tooling applicable to the base element can also be applied to the extension. For example, a well-formedness rule checking that an IOAutomaton has a single initial state can also be applied to the generated RobotADLBehModel. With our approach in general, the right-hand side of the generated production equals the right-hand side of the extended production. If the right-hand side contains a nonterminal that was extended during composition, it is replaced with the newly generated nonterminal name. This transformation is especially necessary for recursive productions and refining in-

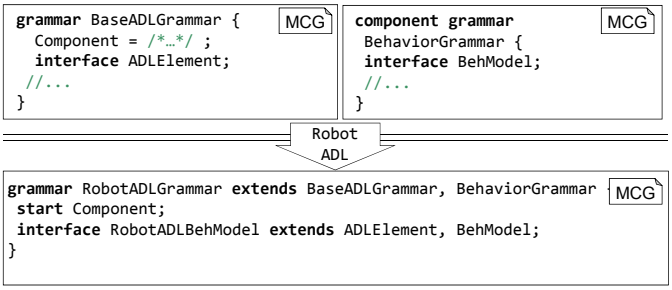


Figure 9: Composition with extension point refinement

terfaces to realize the expected behavior. Without this replacement, parsers would create instances of the abstract syntax class derived from the extension nonterminal instead of the abstract syntax class of the newly generated one. Therefore, the connection between extension point and extension would break, because the extension nonterminal alone has no connection to the extension point. With refinement of extension points, both extension and (as always) extension point are interface productions. Here, a new name is derived in the same style as above in the nonterminal case, but it extends both interfaces. This is visualized in Figure 9.

The language component resulting from the compositions of the two language components `AutomatonLangComp` and `BehaviorLangComp` is depicted in Figure 10. The referenced grammar is the newly generated grammar depicted at the bottom of Figure 8. It combines the abstract syntaxes and concrete syntaxes of the composed grammars. The set of context conditions of the integrated language component is the set union of all sets of context conditions of input language components. Checking all context conditions of the composed language component is possible, because of the compatibility of the abstract syntax tree data structure – that MontiCore relies on for checking context conditions – between the composed grammar and the input grammar that the context condition has been defined for. As no interface productions are removed in the composed grammar, all extension points are joined and the resulting language component comprises all extension points of the input language components. The language component synthesized for a certain language variant can optionally be extended with handwritten grammars and new context conditions before MontiCore is executed to produce tooling for the language variant.

6. Case Study

The following presents an extended case study of our approach in the context of architecture description languages based on the extended example in [8]. Architecture description languages (ADLs) [55] combine MDD with component-based software engineering for the description of software architectures. There are over 120 stand-alone ADLs [54], each tailored towards a specific application domain, such as automotive [16], avionics [24], consumer electronics [81], or robotics [69]. Developing and maintaining domain-specific variants of an ADL is challenging [10].

```
01 language RobotADLLangComp {
02   grammar com.ma.RobotADLGrammar;
03   cocos {
04     com.behavior.SingleBehaviorModel,
05     com.ioaut.SingleInitialState,
06     //...
07   }
08 }
```
LC

Figure 10: The resulting composed language component.

6.1. ADL Language Product Line Motivation and Overview

Our approach enables to prevent the efforts of creating, maintaining, and evolving multiple stand-alone ADL variants tailored to specific domains individually. Using feature-driven language composition enables to start with a core ADL exhibiting extension points and independently developing language components that provide modeling elements required for architectures of the different domains. Modifying one language component automatically updates all language variants that contain the component if the variant derivation is executed again. In the context of ADLs, our approach facilitates to produce tailored ADL variants for new domains, *e.g.,* architectures for machine learning. Common ADL parts can be reused from the language product line, and new features specific to the new domain can easily be added.

Excerpts of a complete language product line for ADLs, called `MyADL`, capable of describing software components for modeling both cloud systems and embedded systems are depicted in Figure 11. Different language engineers contribute language components (depicted right) with explicit extension points defined via their interface productions in the grammar. The language product line engineer defines the feature model and therefore defines which features are available and how selecting a feature may depend on the selection of other features (depicted left). The product line comprises a common base feature (`BaseADL`) and features that are typical to ADLs for embedded systems (*e.g.,* automated connection of ports based on their types or names) as well as features related to scalable and secure cloud systems (*e.g.,* replicating components and encrypted communication). Each feature instantiates exactly one language component that might define further extension points. The relation between two features in the feature model describes how their language components are integrable (parent-child relationship) or whether a feature's language component relies on or conflicts with another language component (cross-tree constraints).

The decoupling between features of the language product line and language components, as well as the decoupling among different language components, enable to reuse the language components in different feature models and to instantiate a language component within different features of a single feature model. Based on a feature configuration defined by the language product owner (middle left), a software tool (implementing the mechanisms described in the previous sections) establishes the connections between the selected features' language

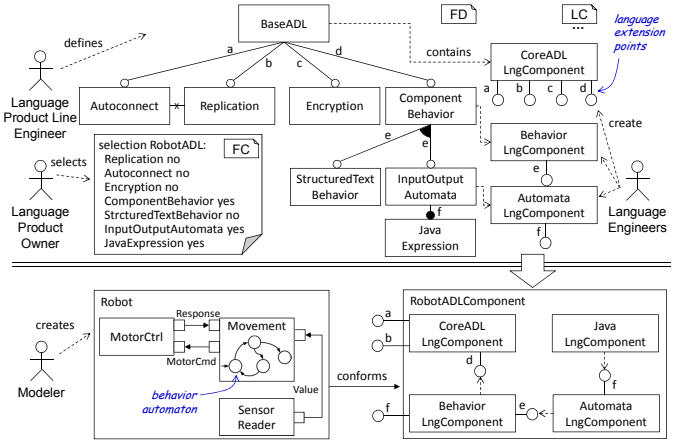


Figure 11: A language product line defined as feature model over language components. Given a feature configuration (top), the variant is transformed into a new language component (bottom).

components. A feature's language component is integrated into the extension points of the language component of the respective parent feature. For instance, the language component of feature `InputOutputAutomata` is integrated into the extension point *e* (the label of the edge between the features `ComponentBehavior` and `InputOutputAutomata` in Figure 11) of the language component contained in the feature `ComponentBehavior`. Integrating all language components of the selected features yields a new language component, which can be used by the respective domain experts to model corresponding software architectures using exactly the modeling elements selected through the feature configuration (in this case automata models describing component behavior).

Being able to reuse language components without modification enables reusing the associated tooling (analyses, transformations) with the generated language component as well. This is possible because the generated AST classes of each individual language component are reused by the tooling of a generated language component. Reuse is possible because the result of the integration of each feature's language component with the language component of the feature's parent stands in a "conservative extension" relationship (in the sense of [67]) with the parent feature's grammar. With this, changes to a language component and its tooling are immediately reflected in all generated language components as well. Therefore, the effort of creating, maintaining, and evolving modeling languages is minimized. The loose coupling between features and language components also simplifies integration of new features into the feature diagram. Integrating a new feature below `ComponentBehavior`, for instance, does not influence other features and language components. The language product line engineer, however, has to ensure potential cross-tree constraints (excludes, implies) of the new feature to existing features.

A generated language component can yield extension points. Thus, the creation of intermediate products that require further refinement is also possible. Where multiple similar domains are addressed, creating refined domain-specific language product lines enables to restrict a large base product line accordingly.

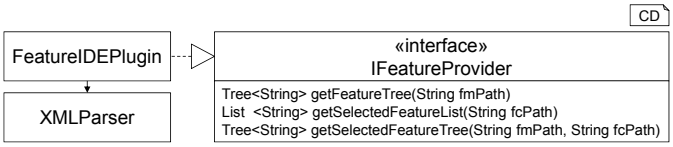


Figure 12: Integration of FeatureIDE [74] into the language composition tool.

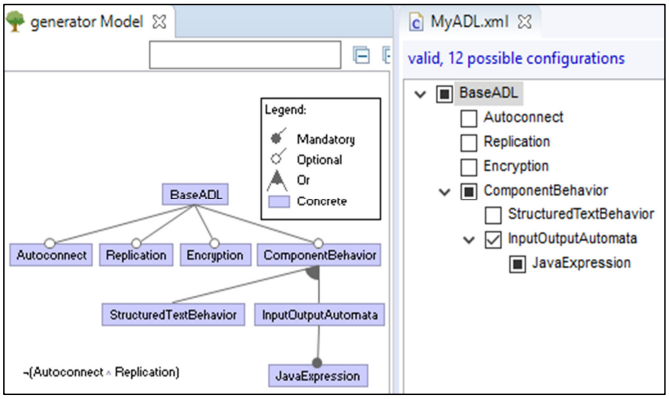


Figure 13: The feature model and the feature configuration in the employed FeatureIDE editors.

The tooling to process language product lines and derive variants is extensible into different directions: it enables to use different languages for the definitions of language components and bindings, and it supports different feature modeling tools. For example, the plug-in integration of FeatureIDE [74] for modeling feature diagrams and feature configurations is depicted in Figure 12. Our language composition tool provides an interface `IFeatureProvider` that has to be implemented by plug-ins for feature modeling tools. The plug-in for FeatureIDE employs the class `FeatureIDEPlugin`, which uses an `XMLParser` to parse the XML artifacts of feature models and feature configurations produced with FeatureIDE.

6.2. ADL Language Product Line Details

The following describes the language product line, its features and language components depicted in Figure 11, and presents a language component derived from a configuration as well as valid and invalid models with regard to the derived language component. The feature model and the selected configuration in FeatureIDE are depicted in Figure 13. The `BaseADL` feature must be domain-agnostic as it is part of all possible product line configurations. It thus only contains the basic elements of ADLs such as components, ports, and connectors that are common to each language variant. The feature `Autoconnect` adds syntax and transformations to realize an automatic connection of ports with either identical names or types. The `Encryption` feature enables describing secure ports (`SecurePort`) and encrypted connections (`EncryptedConnector`) between these. The `Replication` feature provides syntax for modeling systems with components that are capable of replicating themselves if a replication condition is satisfied. This is useful, for instance, in client-server architectures where a client component replicates on a large

```
01 bindings for MyADL {                                          Binding
02   feature BaseADL uses ADLGrammar;
03   feature ComponentBehavior uses ComponentBehaviorGrammar;
04   feature InputOutputAutomata uses IOAutGrammar;
05   feature JavaExpression uses JavaInADLExprGrammar;
06   feature DynamicReconfiguration uses DynReconBehGrammar;
07   feature ModeAutomata uses ModeAutGrammar;
08   //…
09   bind BaseADL.CmpElem to ComponentBehavior.BehModel;
10   bind ComponentBehavior.BehModel to InputOutputAutomata.IOAutomaton;
11   bind InputOutputAutomata.Guard to JavaExpression.GuardExpr;
12   bind InputOutputAutomata.PortAss to JavaExpression.PortAssExpr;
13   bind BaseADL.CmpElem to DynamicReconfiguration.ReconBehModel;
14   bind DynamicReconfiguration.ReconBehModel to ModeAutomata.ModeAut;
15   //…
16 }
```

Figure 14: The binding rules for the language product line.

BaseADL

```
01 grammar ADLGrammar {                                          MCG
02  Component = "component" Name "{" CmpElem* "}";
03  interface CmpElem;
04  interface Port extends CmpElem;
05  DefaultPort implements Port = "port" Type Name ";";
06   /* Subcomponent and Connector productions omitted */
07 }
```

```
01 language CoreADLLngComponent {                                 LC
02  grammar ADLGrammar;
03  cocos {
04   com.adl.cocos.CompNameLowerCase,
05   com.adl.cocos.PortNamesUnique
06  }
07 }
```

Figure 15: Parts of the BaseADL language component.

number of requests. The language product line engineer considers component replication to be a threat for autoconnecting ports. Choosing one of the two corresponding features thus excludes the other. The ComponentBehavior feature (*cf.* Figure 16) introduces *behavior blocks* to the ADL through the interface BehModel. The language product line engineer intends component behavior models to be modeled in such blocks. The child features StructuredTextBehavior and InputOutputAutomata contain different behavior languages. It should not be possible to model empty behavior blocks and thus, choosing the ComponentBehavior feature requires to choose at least one feature that defines a component behavior language. Automata use expressions on their transitions as guard conditions. For this purpose, the language product line currently only includes the JavaExpressions feature, which is therefore marked mandatory.

The *binding* model depicted in Figure 14 describes the mapping from feature name to the name of the language component instantiated by the feature (ll. 2-7) and the binding of extension point of a feature to an extension of another feature (ll. 9-14). The binding model belongs to the language product line MyADL (l. 1). The layer of indirection between feature name and language component name enables to use the same language component within two different features (which must have a unique name within a feature model) of a feature model. The explicit binding between extension point and extension prevents uncontrolled composition (*cf.* Section 4).

ComponentBehavior

```
01 component grammar ComponentBehaviorGrammar {                  MCG
02  interface BehModel;
03 }
```

```
01 language BehaviorLngComponent {                               LC
02  grammar BehaviorGrammar;
03  cocos {
04   com.compbeh.cocos.BehaviorUnique,
05   com.compbeh.cocos.BehaviorNotInComposedComponents
06  }
07 }
```

Figure 16: The ComponentBehavior language feature.

Consider a language product owner who aims at developing software architectures in which atomic components' behavior can be specified via input/output automata. She thus selects the configuration containing the features BaseADL, ComponentBehavior, InputOutputAutomata, and JavaExpression.

InputOutputAutomata

```
01 component grammar IOAutGrammar {                              MCG
02  IOAutomaton = "ioautomaton" "{" AutElem* "}";
03  interface AutElem;
04  State implements AutElem =
05                      (["initial"])? "state" Name ";";
06  Transition implements AutElem = "transition" src:Name
07            "[" Guard "]" "{" PortAss* "}" trg:Name ";";
08  interface Guard;
09  interface PortAss;
```

```
01 language AutLngComponent {                                    LC
02  grammar IOAutGrammar;
03  cocos {
04   com.ioaut.cocos.StateNamesUpperCase,
05   com.ioaut.cocos.UniqueInitialStates
06  }
07 }
```

Figure 17: The InputOutputAutomata language feature.

Figures 15-18 depict parts of the configuration's constituents. The BaseADL language component (*cf.* Figure 15) is the product line's root feature. The feature's grammar defines language elements common to all ADL variants such as components, connector, and ports. The language component further defines two context conditions and the interface CmpElem. With this, it is possible to extend component definitions with further top-level elements through language component composition via binding productions to the interface CmpElem. The Subcomponent and Connector productions of the BaseADL grammar are omitted. The grammar of the ComponentBehavior feature (*cf.* Figure 16) defines a single interface BehModel. Behavior models for atomic components are intended to be embedded into the BehModel interface. The feature's language component further comprises two context conditions. The first ensures that each component contains at most one behavior model. The second requires that composed components must not contain behavior models. The language component binds the BehModel interface to the CmpElem interface of its parent feature's grammar. The language component defines the BehModel interface as extension point. With this, the BehModel extension

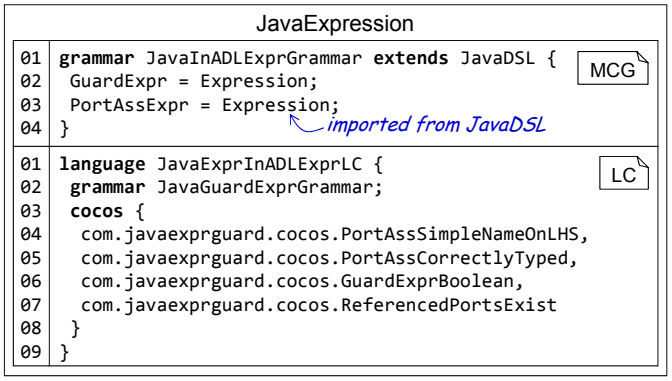


Figure 18: The `JavaExpression` language feature.

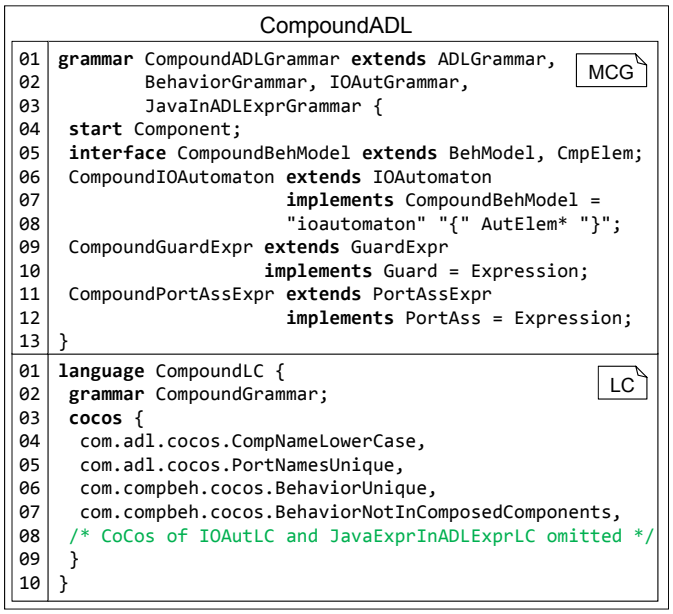


Figure 19: Result of composing the configuration's features.

point refines the `CmpElem` extension point (*cf.* Section 4.2): Every `BehModel` model can be embedded as a `CmpElem`, but the opposite does not hold. Integrating the language components of the `BaseADL` and `ComponentBehavior` features yields a new feature that enables specifying component behavior models as top-level elements in component definitions. However, the syntax of possible component behavior models is still underspecified. For this reason, the `ComponentBehavior` language component is connected to two further features via an `or`-node (*cf.* Figure 11). Thus, each valid configuration containing the `ComponentBehavior` feature also contains at least one of the two child features. The language product owner chooses the `InputOutputAutomata` feature to obtain a valid configuration. The language component's grammar (*cf.* Figure 17) enables to model input/output automata for specifying component behavior. Transitions (ll. 6-7) of such automata consist of guards (l. 8) and port assignments (l. 9). The production's implementations remain underspecified (ll. 8-9). Thus, these are modeled as interfaces by the feature's language component and must be bound by the `InputOutputAutomata` feature's child features. The language component further defines two context conditions, which ensure each input/output automaton contains exactly one initial state and that state names start with capital letters. The grammar's `IOAutomaton` production is embedded into the `BehModel` production of the language component's parent feature. The `JavaExpression` (*cf.* Figure 18) feature is a mandatory child feature of `InputOutputAutomata` and therefore has to be selected by the language product owner. Its grammar inherits the productions from a Java grammar (l. 1) and defines two new productions `GuardExpr` and `PortAssExpr` (ll. 2-3). These enable modeling guards and port assignments with Java expressions. The `Expression` production is part of the inherited Java grammar. The two productions are bound to the `Guard` and `PortAssExpr` interfaces exported by the `InputOutputAutomata` language component. Introducing two new productions in contrast to directly binding the Java `Expression` production to the `Guard` and `PortAssignment` productions enables to separately handle guards and port assignments as they are distinguishable via their types. This is necessary, for instance, if a context condition either only restricts the syntax of port assignments or of guards. The first two context conditions of the feature's language component, *e.g.,* only restrict the well-formedness of expressions used in port assignments (*cf.* Figure 18, ll. 4-5), whereas the third context condition only restricts guard expressions (*cf.* Figure 18, l. 6). The fourth context condition affects guards as well as port assignments. Composing the four features as described in Section 5 yields the language component depicted in Figure 19 that represents the composed language. The grammar is composed of the grammars of the selected features' language components by applying the transformation described in Section 5. The new language component's context conditions are all context conditions of all selected language components. The new language component retains all extension points defined by at least one selected language component. Figure 20 depicts the three models `valid` (ll. 1-8), `invalid1` (ll. 9-17) and `invalid2` (ll. 18-24). The model `valid` is a well-formed model of the new language. The productions for modeling component and port declarations (ll. 1-3) originate from the `ADLGrammar` (*cf.* Figure 15). The `InputOutputAutomata` language component's grammar (*cf.* Figure 17) adds the possibility to declare automata, states, and transitions (ll. 5-7) through extending the interface added by the `ComponentBehavior` language component (*cf.* Figure 16). The expressions `true` and `in = out` used in the transition's guard and port assignment (l. 7) originate from the `JavaExpression` language component (*cf.* Figure 18). The models `invalid1` and `invalid2` are no well-formed models of the new language. Model `invalid1` is an element of the language defined by the new grammar but not well-formed because it violates the three context conditions `PortNamesUnique`, `StateNamesUpperCase`, and `GuardExprBoolean`, which are defined by the three language features `BaseADL`, `InputOutputAutomata`, and `JavaExpression`, respectively. The model `invalid2` is not well-formed because the embedded `ioautomaton` model is not possible in the `In-`

CompoundADL valid

```
component MyComponent {
  port Integer in;
  port Integer out;
  ioautomaton {
    initial state S1; state S2;
    transition S1 [true] {in = out} S1;
  }
}
```

CompoundADL invalid1

```
component MyComponent {
  port Integer in;
  port Integer in; // PortNamesUnique
  ioautomaton {
    initial state s1; // StateNamesUpperCase
    transition S1 [1+1] // GuardExprBoolean
                  {in = out} S1;
  }
}
```

CompoundADL invalid2

```
component MyComponent {
  port Integer in;
  port Integer out;
  ioautomaton {
    init state S1; // parse error
  }
}
```

Figure 20: A valid model and two invalid models of the language component depicted in Figure 19.

`putOutputAutomaton` feature's grammar.

Based on the integrated language component, MontiCore generates model-processing infrastructure to parse models and perform well-formedness checks. Via handwritten extensions, this infrastructure can be further customized.

7. Discussion

The presented approach relies on grammars as descriptions of concrete syntax and abstract syntax. This limits the processable models to be textual. However, building a graphical concrete syntax on top of a textual one is possible [65]. Also, our approach currently only realizes *language embedding* for composition of language components, *i.e.,* a form of composition that produces abstract and concrete syntaxes integrated into a single model. Supporting further forms of language composition, such as language aggregation [67], is subject to further research. The realization of our concept with MontiCore uses *all* interface productions of a grammar as extension points. It is possible to reduce this to a subset of these by explicating the extension point that should be "exported" [8] within a language component model. We introduced this concept to mitigate accidental or unintended extension points when using interface productions for technical reasons. However, our experiences have shown that defining exported extension points explicitly, in practice, was cumbersome and using all interface productions as extension points did not produce problems. To this effect, we omitted the explicit statement of extension points but delegate the decision of when to use interface productions over alternatives to the language (component) engineer. Using disjunctions instead of interface productions can be used to avoid creating unintended extension points. The current realization of our concept does, however, prescribe designers of language components to foresee all extension points by explicating these through interface productions in the grammar. For example in Figure 1, `Condition` and `Action` are the only extension points of the grammar `Automation`. Therefore, extension of `Automation` is restricted to new trigger conditions and actions in the two foreseen places. Technically, MontiCore enables to override any grammar production in order to add new alternatives to their right-hand side [67], which can be leveraged to realize extension points as well. However, we currently do not make use of this to obtain a cleaner abstract syntax and to distinguish which productions are extension points and which are not to foster hiding of internal details of language components. Our concept to syntactic language variability relies on underspecification in the abstract syntax through qualified extension points. These can, for instance, be realized through abstract classes or interfaces in metamodel-based languages [73], through controlled merging of abstract syntax elements [17], or underspecification in grammars, such as binding elements of different languages by name [76].

Designing language components in an appropriate granularity is challenging: If the components are too fine-grained, the feature model becomes complex even for small language product lines. Further, the constituents of languages are scattered across many different language components, which also complicates the understandability and maintainability of these individually. Furthermore, it is unlikely that all language components can be developed independently. However, fine-grained components facilitate the reuse of components in different contexts. Coarse-grained language components, on the other hand, produce feature models that are better readable and reduce scattering of language components, but become more complex. The appropriate granularity is subject to the language engineers. Using a grammar production that is not the start production of the grammar as extension to an extension point cuts off all parts of the language's concrete and abstract syntax that are not reachable as child elements of the abstract syntax induced by the new start production.

Ultimately, our approach flattens the tree structure of the feature model and produces a single composed grammar that directly inherits from the grammars of all selected features and, thereby does not introduce new inheritance relationships between the grammars of the selected features. To this effect, we can allow inheritance dependencies between the grammars contained in different features of a selected variant. Such a dependency should be indicated in the feature model as a *requires* constraint between the feature of the extending grammar and the feature of the extended grammar. As with the composition no additional inheritance relationships are introduced to the grammars of the selected features, inheritance is acyclic if the *requires* constraints in the feature model are acyclic.

Our approach synthesizes a new grammar that integrates the individual grammars of the language components via inheritance. This layer of indirection complicates the readability and understandability of the language syntax. Therefore, the generated grammar is not useful as documentation of the syntax. Through a model-to-model transformation of these grammars, the inheritance relations can easily be flattened to produce a single integrated grammar with improved readability.

The separation of concerns in our approach usually alleviates

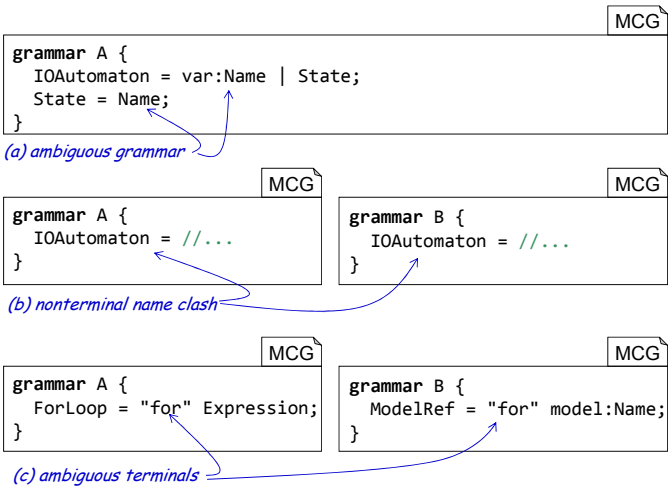


Figure 21: Grammars may be ambiguous (a). Composition of grammars can yield (b) name clashes of nonterminals with the same name and (c) terminals that make parsing ambiguous.

language product owners from being SLE experts and decouples engineering of language components from their composition. In our current work, it is necessary that the language product owner is an SLE expert only if she performs handcrafted, variant-specific customizations. This can be the case, *e.g.,* to customize further tooling such as editors. It is, however, impossible to completely alleviate the language product line engineer, who maintains the composition of language components, from being a software language engineer [78, 77].

MontiCore uses the mechanisms of ANTLR [62] to handle ambiguous grammars, *e.g.,* the grammar depicted in Figure 21 (a). Besides the automatic mechanisms, MontiCore grammars can contain semantic predicates [62] of ANTLR to manually control handling of ambiguities. Furthermore, the composition of independent language components can raise several issues, which are taken care of by the implementation of our approach. As the grammars of different language components typically are developed independent of each other, there might be conflicts in the nonterminals that share the same name unintentionally (*cf.* Figure 21 (b)). With grammar languages that have a hierarchical name space of nonterminals, the qualified names of the nonterminals would differ. For grammar languages with a flat name space (such as the MontiCore grammar language), a transformation of the grammars has to rename conflicting nonterminals to resolve name clashes. As renaming a nonterminal also influences tooling written against this nonterminal, the language composition engineer has to adapt this manually. The realization of our approach based on MontiCore checks whether such name clashes of nonterminals exist and aborts derivation of the language variant on name clashes. Furthermore, the generated parsers may be confronted with ambiguities caused by terminals with the same name that both can occur at a certain place in the model (*cf.* Figure 21 (c)). The parser has to be aware of this, as otherwise two different valid abstract syntax trees could be instantiated from the same model. Currently, our realization with MontiCore aborts generation of a parser if such terminal ambiguities occur. Cross-tree constraints in the feature model restrict valid variants. In our approach, an *excludes* constraint can indicate that the composition causes ambiguities or it indicates that the language product line engineer forbids the composition due to other reasons. This is similar to *requires* constraints, which can either have technical reasons or design intentions. Future work should investigate how to derive such constraints that are due to technical reasons or cause ambiguities from the grammars.

Composition of context conditions can prevent other context conditions to be applicable if the syntactical elements that these operate on are forbidden by another context condition. Further, through extension of the right-hand side of the grammar production of a nonterminal N with a further alternative, a context condition for N also applies to the new alternative. This might be unintentional. For example, the language component `ComponentBehavior` of the case study presented in Section 6 could include a context condition checking that each component behavior declares a name starting with an upper case letter. Later, the language product line is extended with a further language component arranged as child feature of `ComponentBehavior` that enables a different way to specify component behavior. The context condition then also applies to this new language component, which might be unintended. This situation requires to either adapt the context condition to exclude the newly added alternative or to replace it with a new one.

For language product lines with manageable size of involved language components, the approach helps to structure these and the feature model is understandable and well extensible. For large feature models, and language product lines, respectively, it is subject to further research how well the approach scales up.

Through conservative extension implemented by the grammar composition operator, reusability of tooling is increased. If this condition would not hold, tooling written for a certain variant could only guarantee to operate on models of this single language variant. With conservative extension, the set of models that are valid with regard to a language variant A are a subset of the valid models with respect to a language variant B if the set of selected features of A is a subset of the set of selected features of B. As depicted in Figure 22, through conservative extension, tooling of B can be used on models valid in A, because new language syntax can only be integrated by adding further alternatives to existing syntax elements. This property breaks when well-formedness rules are taken into account, for instance, if a well-formedness rule considers those models that do not use a new alternative as invalid. Considering only well-formedness rules (and not the underlying syntax), the relation between valid models A and B can be carried out into the other direction: If through additional language components in the language variants no existing rules can be eliminated, only new rules can be added. As with each new rule, the set of valid models can only decrease in size, valid models of B are a subset of valid models in A. However, well-formedness rules are checked after parsing and therefore, each activity realized before well-formedness checking benefits from the conservative extension. With subsequent activities in the model-processing pipeline, such as interpretation or code generation, the prop-

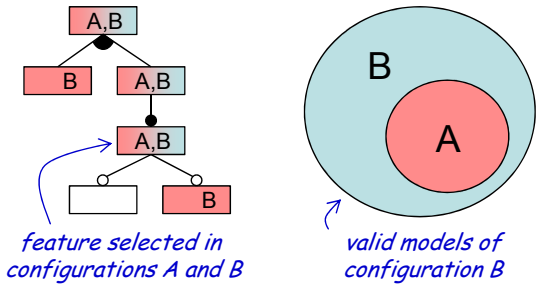


Figure 22: If the set of features contained in configuration *A* is a subset of the features contained in configuration *B*, the valid models of the language variant derived from *A* are a subset of valid models of the language variant *B*.

erty of conservative extension for language components does not hold anymore. With parsers, well-formedness checks, and further analyses and transformations that operate on the abstract syntax of (potentially ill-formed) models, the conservative extension property ensures proper reusability.

The presented approach has no explicit types for different kinds of language components. Besides that, it is questionable in which dimension typing languages should be carried out (*e.g.,* imperative or declarative languages, expression language or behavior language, language with code generators translating to the same language, ...), it could limit reusability of a language component for a different purpose. The advantage of a type system of language components is obvious: if many language components are available for similar purposes, typing these would limit the choice between finding a suitable one. Further, typing could reduce the knowledge about a language component's content required by language product line engineers. Instead of an explicit type system for language components, the extension points defined in a language component make assumptions about the language components that can be employed to deliver possible extensions.

Many other approaches imply that either the feature model is developed before the assets (top-down approach) or the assets are available and the feature model is built or derived for existing assets (bottom-up approach) [49]. Due to the loose coupling between the feature model and the language components of our approach, both paradigms are supported. While in our current work, we focus on a bottom-up approach, future work should investigate developing a language product line top-down.

The realization of our approach leverages MontiCore grammars as integrated descriptions of the concrete and abstract syntax of language components. Due to the inherent coupling between these in being defined within the same grammar rules, the realization does not support pure presentational variability [11], *i.e.,* variability within the concrete syntax only.

Another challenge arises from composing not only syntax but also the languages' behavior realizations (if available), which usually have the form of code generators [5, 14] or interpreters [6, 80]. Composition of both has been achieved for specific languages [64, 66] or under various restrictions [4, 40]. We presented a first approach for an integration of code generator composition into our concept for language product lines in [9]. However, this concept has limitations that have to be mitigated and a general approach towards code generator composition yet remains to be conceived.

According to [43], a good language extension framework must support independent language extensions and automated composition of these. Our approach supports both, but is even more powerful than a language extension framework, as it leaves the choice of a base language open to the language product line engineers, instead of being restricted to a fixed base language. Further, [43] states that composition must not produce a corrupted compiler. As with the transformations of conflicting grammar productions described above, and the conservative extension property, our approach guarantees to produce either an uncorrupted parser or no parser. With context conditions and the translation of a compiler (which would be carried out to MontiCore code generators in the realization of our approach), we cannot guarantee this anymore. Future work should investigate extracting language product lines from existing handcrafted, cloned-and-owned language variants.

8. Related Work

Variability within software languages has been investigated in a wealth of different approaches [56]. Modeling languages are usually engineered by means of language workbenches, for which different techniques of reusing and composing languages exist [21]. Further, language workbenches differ in the language constituents they process and the constituents of the languages they produce. Several language workbenches, including Rascal [79], MontiCore [30], Neverlang [76], Spoofax [45], and Xtext [5] define syntax via grammars. Other language workbenches employ metamodels for syntax definitions, including EMF [73], GEMOC Studio [17], or MetaEdit+ [75], or employ projectional editors, such as MPS [89]. Well-formedness rules are usually either implemented with OCL [32] or as GPL statements [30]. The following considers related work for each research question (*cf.* Section 4) individually.

RQ1: *Variability within language module*
With language definitions that employ metamodels for syntax definitions, concrete syntax often plays a minor role. To this effect, metamodel-based approaches typically support variability in abstract syntax only [38, 63, 90] while grammar-based approaches often consider concrete syntax and abstract syntax [53, 57, 76]. The language development framework Neverlang [76] supports modular language definitions. A language module comprises a grammar as syntax definition and a corresponding ordered list of evaluation phases. These evaluation phases realize the semantics of a language module and include type checking, well-formedness checking, and code generation. Extension points in Neverlang grammars are realized as placeholders, which are unused nonterminal names on the right-hand sides of grammar productions. Compared to our approach, these cannot prescribe the presence of certain abstract syntactical elements, enabling easier reusability, but bearing higher complexity in finding a module providing a suitable extension.

Further, explicating extension point through, *e.g.,* interface nonterminals reduces the risk of defining extension points unintendedly (*e.g.,* by misspelling a nonterminal).

The revisitor approach [52, 51] uses Ecore metamodels for describing the syntax of language modules. It enables to describe variability within a language's metamodel and the realization of its semantics by using the revisitor pattern. Extension points in a metamodel are realized as required metamodel elements.

The set of languages mbeddr [84, 85] is built upon MPS [83] and thereby, uses its projectional editors and further sophisticated tooling. In MPS, abstract syntax elements can extend other abstract syntax elements to perform language extension. Therefore, every abstract syntax element is a potential extension point.

Action-semantics modules [18] leverage context-free grammars to describe the syntax of such modules. To realize extension points, context-free grammars can define unused nonterminals. LISA [57] uses attribute grammars to describe the concrete syntax, abstract syntax, and semantics of language modules. LISA uses inheritance as known from object-oriented programming to realize language inheritance on attribute grammars. In principle, every rule of the grammar can be extended by new rules. The combination of SDF and FeatureHouse realizes compositional language modules [53] containing grammar rules, typing rules, and evaluation rules. Variability is carried out into two dimensions: the dimension of extension with new language concepts and the dimension of extension with new tooling (*e.g.,* new typing rules). For realizing the variability in the grammar rules, the SDF modularization is used and Spoofax [45] is employed to generate parsers. Silver [93] is an extensible attribute grammar system. It uses attribute grammars modules whose syntax and semantics can be extended.

RQ2: *Composition of language modules*
Our approach uses composition to derive a language variant, and therefore the related work focuses on compositional approaches as well. However, there are approaches that define 150% metamodels and use negative variability to reduce these it to obtain language variants [90].

Neverlang supports several forms of composition for the syntax of language components [76]. Language extension relies on composition of the grammars, where one component provides all implementations that another component requires. Language unification employs glue code to match required and provided implementations that do not match originally. The approach for development of languages with action semantics modules [18] relies on composition of the grammar productions for composing syntaxes. In this, the unused nonterminals serving as extension points can be implemented by importing language modules that realize these unused nonterminals. To the best of our knowledge, the approach does not have a mechanism to compose independent language modules for building language product lines with an explicit variability model.

SugarJ [20] enables to specify syntactic extensions to Java. These extensions are contained in syntactic sugar libraries. By "desugaring", the extended syntax is transformed into the base syntax. SugarJ uses parsers that are capable of detecting ambiguities, on which they report an error.

The language framework ableC [42] is an extensible C language implemented in Silver [93] that uses Copper as parser generator. ableC uses attribute grammars for describing the syntax of independent language extensions and provides different composition mechanisms for these extensions. The mechanism satisfies several criteria that make it expressive enough to provide a solution to the expression problem. Their reliable and automatic composition mechanism guarantees correct composition of attribute grammars and, therefore, also related analyses and transformations. The base language C, however, cannot be exchanged. Silver [93] uses an "import with syntax" mechanism to compose the syntax of attribute grammars. The semantics of the import is that rules of imported grammars are as if they were specified in the importing grammar.

Wyvern [61] is a further extensible language system that guarantees reliable composition. It uses delimiters (*e.g.,* braces, whitespace) to coordinate parsing between base language fragments and language extension fragments. Our approach investigates language composition on a more abstract level as the above-mentioned approaches. It is based on MontiCore that itself generates parsers by employing ANTLR [62]. The detected ambiguities are therefore limited to those directly accessible in the grammar (*cf.* Section 7).

In the revisitor approach [52], extension points do not have to be foreseen by language developers at design time and composition leverages the revisitor pattern. Further, the several extensions can be independent of each other. A binding [51] realizes an adapter functionality by connecting two metamodel elements of different metamodels. Mbeddr [84] includes an extensible set of language modules that have C as their base language. Therefore, it does not support to use a different base language than C, which limits its applicability for, *e.g.,* pure model-driven development. mbeddr supports three types of language composition: language modules can be loosely coupled by remaining in separate artifacts with language referencing. In this form of composition, a language module references a part of another language module only. Embedding realizes a syntactic composition between independent language modules similar to our approach. Language extension is realized by a language module extending the syntax of a host language module, which it must be aware of. mbeddr mostly relies on language extension to realize composition of language modules. LISA [53] uses an inheritance mechanism to compose attribute grammars. SDF+FeatureHouse [53] employs techniques of FeatureHouse to compose language modules with superimposition, weaving, or inheritance as composition operators.

RQ3: *Meaningful combinations of language modules*
Copper uses modular analyses [70] to verify that the composition of grammars will result in a valid grammar and a deterministic parser. These analyses are carried out independently on each extension to a base grammar and, if they are fulfilled, guarantee several properties.

Several approaches propose to employ feature diagrams to arrange language modules in a form that restricts possible com-

binations [37, 49, 50, 53, 78, 90]. There is an extension to Neverlang [78] that uses the common variability language to organize language modules and their interrelations. Another extension, AiDE [50], builds upon Neverlang and derives variability models from an existing landscape of interrelated Neverlang modules. The addressed use case of this is to post-hoc derive a language product line from an existing set of language modules used for a specific purpose. To do so, AiDE first extracts all dependencies between employed input language modules and then synthesizes a feature model. AiDE can be leveraged to describe language product lines but is not capable of developing language modules independently, and then build up a language product line of these.

To the best of our knowledge, the organization of language modules in mbeddr [84], LISA [57], action-semantics modules [18], the revisitor approach [51], and ableC [42] do not support building dedicated language product lines with controlled arrangement and interrelations between employed language modules.

RQ4: *Separation of concerns*
A separation of concerns between different actors or roles that are involved in developing a language product line, as described in Section 4.3 is only described for some of the approaches. Neverlang [78] distinguishes between language developers and domain experts who are involved in the process of developing language product lines. While language developers create language.

Wyvern [61] and Copper [92] completely alleviate composition engineer from understanding the details of the components. in mbeddr [84], any language engineer familiar with MPS can create language extensions. The decision which language modules should be developed and when these are engineered should be coordinated for rather central extensions used by many people of, *e.g.,* an organization. For small extensions, this is less relevant. In Argyle [37], DSLs are constructed from language assets. A feature model is created during domain analysis. The DSL user is a programmer and selects necessary functions to fulfill requirements of the target DSL.

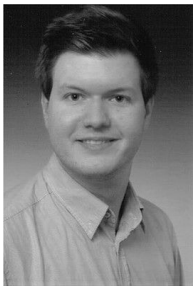
9. Conclusion

We presented a concept to engineer and maintain syntactic language features independent of another within language components. The language components inherit the extension points of the grammar they contain and enable controllable and systematic composition through these. This facilitates engineering new languages by reusing existing components instead of recreating their concepts and related artifacts from scratch. To guide composition of language components and liberate language engineers and modelers from comprehending the internals of all participating components, we propose to relate the language components through feature models. This enables defining product lines of languages that can be arranged by dedicated language product line engineers, which ensure that only valid (in a subjective sense) language products can be derived. Based on a feature configuration, language product owners can easily compose a new language from existing components without language expertise. This, ultimately, facilitates the engineering, maintenance, and evolution of software languages.

Vitae

Arvid Butting received his B. Sc. and M. Sc. degrees in computer science from the RWTH Aachen University, in 2014 and 2016. Currently, he is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. His research interests cover software language engineering, software architectures, and model-driven development.

Robert Eikermann received his B. Sc. and M. Sc. degrees in computer science from the RWTH Aachen University, in 2012 and 2014. Currently, he is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. His research interests cover software language engineering, behavior languages, and model-driven software development.



Oliver Kautz received his B. Sc. and M. Sc. degrees in computer science from the RWTH Aachen University, in 2014 and 2016. Currently, he is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. His research interests cover software engineering, software language engineering, software architectures, model-driven software development, and modeling language semantics.

Bernhard Rumpe is chair of the Department for Software Engineering at the RWTH Aachen University, Germany. His main interests are software development methods and techniques that benefit from both rigorous and practical approaches. This includes the impact of new technologies such as model-engineering based on UML-like notations and domain-specific languages and evolutionary, test-based methods, software architecture as well as the methodical and technical implications of their use in industry. He has furthermore contributed to the communities of formal methods and UML. Since 2009 he started combining modeling techniques and cloud computing. He is author and editor of eight books and editor-in- chief of the Springer International Journal on Software and Systems Modeling. See `http://www.se-rwth.de/topics/` for more.

Andreas Wortmann received his Ph.D. from RWTH Aachen University in 2016. Currently, he is a tenured researcher at the Department for Software Engineering at RWTH Aachen University. His research interests cover software engineering, software language engineering, model-driven development, and robotics. He is a member of IEEE and its Technical Committee on Software Engineering for Robotics and Automation and serves on the board of the European Association for Programming Languages and Systems (EAPLS).

References

[1] Kai Adam, Arvid Butting, Oliver Kautz, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Type-safe Interfaces into Template-based Code Generators. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18)*, pages 179 – 190. SciTePress, January 2018.

[2] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, sep 2003.

[3] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005.

[4] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca Model of Software-System Generators. *IEEE Software*, 11(5):89–94, September 1994.

[5] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[6] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution framework of the gemoc studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 84–89. ACM, 2016.

[7] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)*, pages 187–199. ACM, 2018.

[8] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18)*, pages 75–82. ACM, January 2018.

[9] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

[10] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *Modelling Foundations and Applications (ECMFA'17), Held as Part of STAF 2017*, pages 53–70. Springer International Publishing, 2017.

[11] María Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within modeling language definitions. *Model Driven Engineering Languages and Systems*, pages 670–684, 2009.

[12] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[13] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures*, 54:139 – 155, 2018.

[14] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[15] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Using Feature Models. In *International Conference on Software Product Lines*, pages 266–283. Springer, 2004.

[16] Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. An Architecture Description Language. In *Architecture Description Languages*, pages 181–195. Springer, 2005.

[17] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)*, Pittsburgh, United States, 2015.

[18] Kyung-Goo Doh and Peter D Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36, 2003.

[19] Matthew Emerson and Janos Sztipanovits. Techniques for Metamodel Composition. In *OOPSLA–6th Workshop on Domain Specific Modeling*, pages 123–139, 2006.

[20] Sebastian Erdweg, Lennart CL Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Library-based Model-driven Software Development with SugarJ. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 17–18. ACM, 2011.

[21] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D.P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches. In *Software Language Engineering*. Springer International Publishing, 2013.

[22] Moritz Eysholdt and Heiko Behrens. Xtext - Implement your Language Faster than the Quick and Dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM.

[23] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical Language Analysis in Software Linguistics. In *SLE*, pages 316–326. Springer, 2010.

[24] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012.

[25] Charles Forsythe. *Instant FreeMarker Starter*. Packt Publishing Ltd, 2013.

[26] Robert France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE.*, 2007.

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[28] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System model-based definition of modeling language semantics. In *FMOODS/FORTE*, pages 152–166, 2009.

[29] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[30] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Voelkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, Angers, France, 2015. Scitepress.

[31] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[32] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. Integrating ocl and textual modelling languages. In *International Conference on Model Driven Engineering Languages and Systems*, pages 349–363. Springer, 2010.

[33] Robert Heim, Pedram Mir Seyed Nazari, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Modeling Robot and World Interfaces for Reusable Tasks. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1793–1798, 2015.

[34] C. A. R. Hoare. Hints on Programming Language Design. Technical report, Stanford University, Stanford, CA, USA, 1973.

[35] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, 54:386 – 405, 2018.

[36] Andreas Horst and Bernhard Rumpe. Towards Compositional Domain Specific Languages. In *Proceedings of the 7th Workshop on Multi-Paradigm Modeling (MPM'13)*, pages 1–5. Citeseer, 2013.

[37] C. Huang, A. Osaka, Y. Kamei, and N. Ubayashi. Automated dsl construction based on software product lines. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 1–8, 2015.

[38] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, 14(2):905–920, 2015.

[39] He Jifeng, Xiaoshan Li, and Zhiming Liu. Component-Based Software Engineering. In *International Colloquium on Theoretical Aspects of Computing*, pages 70–95. Springer, 2005.

[40] Sven Jörges. *Construction and evolution of code generators: A model-driven and service-oriented approach*, volume 7747. Springer, 2013.

[41] Frédéric Jouault and Jean Bézivin. Km3: a dsl for metamodel specification. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer, 2006.

[42] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework. *Proc. ACM Program. Lang.*, 1(OOPSLA):98:1–98:29, October 2017.

[43] Ted Kaminski and Eric Van Wyk. Creating and using domain-specific language features. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, pages 18–21. ACM, 2013.

[44] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[45] Lennart CL Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *ACM sigplan notices*, volume 45, pages 444–463. ACM, 2010.

[46] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages using Metamodels*. Pearson Education, 2008.

[47] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Softwware Engineering Methodology*, 14(3):331–380, July 2005.

[48] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. In *International Journal on Software Tools for Technology Transfer (STTT)*, 2010.

[49] Thomas Kühn and Walter Cazzola. Apples and oranges: comparing top-down and bottom-up language product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 50–59. ACM, 2016.

[50] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. Choosy and Picky: Configuration of Language Product Lines. In *Proceedings of the 19th International Software Product Line Conference*, pages 71–80. ACM, 2015.

[51] Manuel Leduc, Thomas Degueule, and Benoit Combemale. Modular Language Composition for the Masses. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, pages 47–59. ACM, 2018.

[52] Manuel Leduc, Thomas Degueule, Benoît Combemale, Tijs Van Der Storm, and Olivier Barais. Revisiting Visitors for Modular Extension of Executable DSMLs. In *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*, Austin, United States, September 2017.

[53] Jörg Liebig, Rolf Daniel, and Sven Apel. Feature-oriented Language Families: A Case Study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 11:1–11:8, New York, NY, USA, 2013. ACM.

[54] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 2013.

[55] Nenad Medvidovic and Richard N Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 2000.

[56] David Méndez-Acuña, José A Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. Leveraging software product lines engineering in the development of external dsls: A systematic literature review. *Computer Languages, Systems & Structures*, 46:206–235, 2016.

[57] Marjan Mernik. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software*, 86(9), 2013.

[58] Brice Morin, Gilles Perrouin, Philippe Lahire, Olivier Barais, Gilles Vanwormhoudt, and Jean-Marc Jézéquel. Weaving variability into domain metamodels. *Model driven engineering languages and systems*, pages 690–705, 2009.

[59] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*, 1969.

[60] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In I. Ober, A. S. Gokhale, J. H. Hill, J.-M. Bruel, M. Felderer, D. Lugato, and A. Dabholka, editors, *Proceedings of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud Computing*, volume 1118 of *CEUR*, pages 15–24, Miami, Florida, USA, 2013. CEUR-WS.org.

[61] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely Composable Type-Specific Languages. In *European Conference on Object-Oriented Programming*, pages 105–130. Springer, 2014.

[62] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[63] Luis Pedro, Matteo Risoldi, Didier Buchs, Bruno Barroca, and Vasco Amaral. Composing visual syntax for domain specific languages. *Human-Computer Interaction. Novel Interaction Methods and Techniques*, pages 889–898, 2009.

[64] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. In *1st International Workshop on Model-Driven Robot Software Engineering (MORSE 2014)*, volume 1319 of *CEUR Workshop Proceedings*, pages 66 – 77, York, Great Britain, July 2014.

[65] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In Stefan Wagner and Horst Lichter, editor, *Software Engineering 2013 Workshopband*, volume 215 of *LNI*, pages 155–170. GI, Köllen Druck+Verlag GmbH, Bonn, 2013.

[66] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, 2016.

[67] Bernhard Rumpe and Katrin Hölldobler. *MontiCore 5 Language Workbench. Edition 2017*. Aachener Informatik-Berichte, Software Engineering Band 32. Shaker Verlag, 2017.

[68] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[69] Christian Schlegel, Andreas Steck, and Alex Lotz. Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model. In *Introduction to Modern Robotics*. iConcept Press, 2011.

[70] August C Schwerdfeger and Eric R Van Wyk. Verifiable Composition of Deterministic Grammars. *ACM Sigplan Notices*, 44(6):199–210, 2009.

[71] Friedrich Steimann, Marcus Frenkel, and Markus Völter. Robust Projectional Editing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 79–90. ACM, 2017.

[72] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2008.

[73] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[74] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 79:70–85, 2014.

[75] Juha-Pekka Tolvanen and Steven Kelly. MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 819–820. ACM, 2009.

[76] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.

[77] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. Automating Variability Model Inference for Component-Based Language Implementations. In *Proceedings of the 18th International Software Prod-*

uct Line Conference, pages 167–176. ACM, 2014.

[78] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. Variability support in domain-specific language development. In *International Conference on Software Language Engineering*, pages 76–95. Springer, 2013.

[79] Tijs van der Storm. *The Rascal Language Workbench*. CWI. Software Engineering [SEN], 2011.

[80] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[81] Rob Van Ommering, Frank Van Der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.

[82] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. Sirius: A rapid development of DSM graphical editor. In *Intelligent Engineering Systems (INES), 2014 18th International Conference on*, pages 233–238. IEEE, 2014.

[83] Markus Voelter and Vaclav Pech. Language modularity with the mps language workbench. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1449–1450. IEEE, 2012.

[84] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140. ACM, 2012.

[85] Markus Voelter, Jos Warmer, and Bernd Kolb. Projecting a Modular Future. *IEEE Software*, 32(5):46–52, 2015.

[86] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. 2011. Shaker Verlag, 2011.

[87] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C L Kats, Eelco Visser, and Guido Wachsmuth. {*DSL*} *Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[88] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. Wiley, 2013.

[89] Markus Völter and Eelco Visser. Language extension and composition with language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–304. ACM, 2010.

[90] Jules White, James H Hill, Jeff Gray, Sumant Tambe, Aniruddha S Gokhale, and Douglas C Schmidt. Improving domain-specific language reuse with software product line techniques. *IEEE software*, 26(4), 2009.

[91] Andreas Wortmann, Benoit Combemale, and Olivier Barais. A Systematic Mapping Study on Modeling for Industry 4.0. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 281–291. IEEE, September 2017.

[92] Eric R. Van Wyk and August C. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 63–72, New York, NY, USA, 2007. ACM.

[93] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an Extensible Attribute Grammar System. *Electronic Notes in Theoretical Computer Science*, 2008.

[94] Steffen Zschaler, Dimitrios S Kolovos, Nikolaos Drivalos, Richard F Paige, and Awais Rashid. Domain-specific metamodelling languages for software language engineering. In *International Conference on Software Language Engineering*, pages 334–353. Springer, 2009.